Last Class: OS and Computer Architecture

- CPU, memory, I/O devices, network card, bus
Last Class: OS and Computer Architecture

<table>
<thead>
<tr>
<th>OS Service</th>
<th>Hardware Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection</td>
<td>Kernel/user mode, protected instructions, base/limit</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Interrupt vectors</td>
</tr>
<tr>
<td>System calls</td>
<td>Trap instructions and trap vectors</td>
</tr>
<tr>
<td>I/O</td>
<td>Interrupts and memory</td>
</tr>
<tr>
<td>Scheduling, error recovery, accounting</td>
<td>Timer</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Atomic instructions</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>Translation look-aside buffers</td>
</tr>
</tbody>
</table>
Clicker Question #1

When a user-level program wants to call a privileged assembly instruction, it:

(A) Signals another user-level process, running as root, to execute the instruction.

(B) Traps to the OS via a system call, who then executes the instruction on its behalf.

(C) It causes an interrupt that signals the CPU to execute the assembly code.

(D) None of the above
Answer on Next Slide
OS Structures & Services

• More on System Calls

• Introduce the organization and components in an OS.

• **Four example OS organizations**
  
  – Monolithic kernel
  
  – Layered architecture
  
  – Microkernel
  
  – Modular
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?
Standard C Library Example

- C program invoking printf() library call, which calls write() system call
Example of Standard API

• Consider the ReadFile() function in the Win32 API—a function for reading from a file

A description of the parameters passed to ReadFile()

- HANDLE file—the file to be read
- LPVOID buffer—a buffer where the data will be read into and written from
- DWORD bytesToRead—the number of bytes to be read into the buffer
- LPDWORD bytesRead—the number of bytes read during the last read
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used
System Call Implementation

• Typically, a number associated with each system call
  – System-call interface maintains a table indexed according to these numbers

• The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values

• The caller need know nothing about how the system call is implemented
  – Just needs to obey API and understand what OS will do as a result call
  – Most details of OS interface hidden from programmer by API

• Managed by run-time support library (set of functions built into libraries included with compiler)
System Call: OS Relationship

user application

open ()

user mode

system call interface

kernel mode

open ()

Implementation of open () system call

return
Sys Call Parameter Passing

• Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call

• Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    • In some cases, may not fit in registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    • This approach taken by Linux and Solaris
  - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed
# Writes "Hello, World" to the console using a system call. Runs on 64-bit Linux
# To assemble and run:
#     gcc -nostdlib hello.s && ./a.out

.global _start

.text
_start:

# write(1, message, 13)
mov     $1, %rax                # system call 1 is write
mov     $1, %rdi                # file handle 1 is stdout
mov     $message, %rsi          # address of string to output
mov     $13, %rdx               # number of bytes
syscall                         # invoke operating system to do the write

# exit(0)
mov     $60, %rax               # system call 60 is exit
xor     %rdi, %rdi              # we want return code 0
syscall                         # invoke operating system to exit

message:
    .ascii  "Hello, world\n"
Clicker Question #2

What is the assembly instruction for a system call on x86 64bit?

(A) syscall

(B) int

(C) mov

(D) halt and catch fire
Answer on Next Slide
Examples of System Calls

<table>
<thead>
<tr>
<th>Category</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Control</td>
<td>CreateProcess()</td>
<td>fork()</td>
</tr>
<tr>
<td></td>
<td>ExitProcess()</td>
<td>exit()</td>
</tr>
<tr>
<td></td>
<td>WaitForSingleObject()</td>
<td>wait()</td>
</tr>
<tr>
<td>File Manipulation</td>
<td>CreateFile()</td>
<td>open()</td>
</tr>
<tr>
<td></td>
<td>ReadFile()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>WriteFile()</td>
<td>write()</td>
</tr>
<tr>
<td></td>
<td>CloseHandle()</td>
<td>close()</td>
</tr>
<tr>
<td>Device Manipulation</td>
<td>SetConsoleMode()</td>
<td>ioctl()</td>
</tr>
<tr>
<td></td>
<td>ReadConsole()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>WriteConsole()</td>
<td>write()</td>
</tr>
<tr>
<td>Information</td>
<td>GetCurrentProcessID()</td>
<td>getpid()</td>
</tr>
<tr>
<td>Maintenance</td>
<td>SetTimer()</td>
<td>alarm()</td>
</tr>
<tr>
<td></td>
<td>Sleep()</td>
<td>sleep()</td>
</tr>
<tr>
<td>Communication</td>
<td>CreatePipe()</td>
<td>pipe()</td>
</tr>
<tr>
<td></td>
<td>CreateFileMapping()</td>
<td>shmmget()</td>
</tr>
<tr>
<td></td>
<td>MapViewOfFile()</td>
<td>mmap()</td>
</tr>
<tr>
<td>Protection</td>
<td>SetFileSecurity()</td>
<td>chmod()</td>
</tr>
<tr>
<td></td>
<td>InitializeSecurityDescriptor()</td>
<td>umask()</td>
</tr>
<tr>
<td></td>
<td>SetSecurityDescriptorGroup()</td>
<td>chown()</td>
</tr>
</tbody>
</table>
One Basic OS Structure

- The *kernel* is the protected part of the OS that runs in kernel mode, protecting the critical OS data structures and device registers from user programs.

- Debate about what functionality goes into the kernel (above figure: UNIX) - “monolithic kernels”
Mac OS X Architecture
Layered OS design

*Layer N:* uses layer N-1 and provides new functionality to N+1

- Advantages: modularity, simplicity, portability, ease of design/debugging

- Disadvantage - communication overhead between layers, extra copying, book-keeping
Microkernel

- Small kernel that provides communication (message passing) and other basic functionality
- other OS functionality implemented as user-space processes
Microkernel Features

- **Goal**: to minimize what goes in the kernel (mechanism, no policy), implementing as much of the OS in User-Level processes as possible.

- Advantages
  - better reliability, easier extension and customization
  - mediocre performance (unfortunately)

- First Microkernel was Hydra (CMU '70). Current systems include Chorus (France) and Mach (CMU).
Mac OS X - hybrid approach

- Layered system: Mach microkernel (mem, RPC, IPC) + BSD (threads, CLI, networking, filesystem) + user-level services (GUI)
Modules

• Most modern operating systems implement kernel modules
  – Uses object-oriented approach
  – Each core component is separate
  – Each talks to the others over known interfaces
  – Each is loadable as needed within the kernel

• Overall, similar to layers but with more flexible

• lsmod on Linux will show you the modules
Clicker Question #3

In modern OSs, file systems are typically part of:

(A) each program

(B) user-level processes that provide services to other programs

(C) the os kernel

(D) the hard disk driver

(E) the hard disk controller
Answer on Next Slide
Processes

- The OS manages a variety of activities:
  - User programs
  - Batch jobs and command scripts
  - System programs: printers, spoolers, name servers, file servers, network listeners, etc.

- Each of these activities is encapsulated in a process.

- A process includes the execution context (PC, registers, VM, resources, etc.) and all the other information the activity needs to run.

- A process is not a program. A process is one instance of a program in execution. Many processes can be running the same program. Processes are independent entities.
OS and Processes

• The OS creates, deletes, suspends, and resumes processes.

• The OS schedules and manages processes.

• The OS manages inter-process communication and synchronization.

• The OS allocates resources to processes.
What's in a Process?

- **Process**: dynamic execution context of an executing program

- Several processes may run the same program code, but each is a distinct process with its own state

- A process executes sequentially, one instruction at a time
Process State

- **Process state** consists of at least:
  - the code for the running program,
  - the static data for the running program,
  - space for dynamic data (the heap), the heap pointer (HP),
  - the Program Counter (PC), indicating the next instruction,
  - an execution stack with the program's call chain (the stack), the stack pointer (SP)
  - values of CPU registers
  - a set of OS resources in use (e.g., open files)
  - process execution state (ready, running, etc.).
Process State in Memory

What you wrote:

```c
void X (int b){
    if ( b == 1 ) ...
}
main(){
    int a = 2;
    X ( a );
}
```

What's in memory:

- **Static Data Segment**: `main; a=2`
- **Heap**: `X; b = 2`
- **Stack**:
  - `void X ( int b ) {
    if (b==1)...
  }
  - `void main() {
    int a = 2
    X ( a );
  }
`
Process Execution State

- Execution state of a process indicates what it is doing
  
  *new*: the OS is setting up the process state
  
  *running*: executing instructions on the CPU
  
  *ready*: ready to run, but waiting for the CPU
  
  *waiting*: waiting for an event to complete
  
  *terminated*: the OS is destroying this process

- As the program executes, it moves from state to state, as a result of the program actions (e.g., system calls), OS actions (scheduling), and external actions (interrupts).
Process Execution State

Example:

```c
void main() {
    printf('Hello World');
}
```

- The OS manages multiple active process using state queues (More on this later…)